第三届阿里中间件性能挑战赛

挑战双十一万亿级消息引擎

阿里中间件 ALI middleware　TIANCHI天池

Rapids团队
香港科技大学
车煜林 王立鹏 赖卓航

内容

1、赛题理解、核心思路概览

2、重放计算流水线、网络传输落盘设计

3、工程价值、通用性分析

4、理论最优时间、tricks分析

5、版本演进、比赛总结

赛题要求 - 数据库主从增量同步

- 输入，在Server端
  - 顺序append日志文本的文件10个，总大小10GB
  - 只能单线程，顺序读取文件1次，模拟真实流式处理场景
  - LogOperation(数据操作类型)：insert key(带有全部属性值)，delete key，update property(单个属性)，update key

- 输出，在Client端
  - 数据库重放后，把在查询范围内，按主键排序的记录行(按照插入时候顺序，排列属性值)，落盘在client端

- 比赛环境
  - 两台配置相同的，16逻辑CPU核的虚拟机
  - JVM堆上限3GB，Direct Memory上限200MB

赛题理解 – 单库单表重放，范围查询

- 日志数据
  - 单表日志，数据库初始为空
  - 通过第一条日志可获取单表的字段信息
  - 表字段的长度有确定范围，选手可以基于此进行优化

- 查询范围
  - 单库单表查询
  - range在(1,000,000, 8,000,000)，重放后符合范围的记录条数占查询区间最大条数比约1/7
  - 输出文件大小大约38MB，网络带宽不会成为系统瓶颈。

赛题要点 – 处理并发和同步，优化内存使用

- 评测环境特点
  - 多线程环境，需解决并发同步问题，利用好线程级并行
  - 输入10GB文件在ramfs，顺序单线程访问所需理论时间2.5s，要设计好计算流水线，来overlap计算和IO

- 设计和优化要点
  - 并行处理字节块到LogOperation(数据操作对象)的解析，处理重放计算时需保证的LogOperation[]顺序性的同步问题
  - 并行产生重放后范围内记录行对应的byte[]
  - 文件读取采用mmap，网络传输和落盘采用nio的zero-copy方式，减少内核态和用户态拷贝
  - 优化内存使用，注意java小对象的extends Object和内存对齐开销，hashmap和hashset使用内存友好的开地址设计并采用高效的probing方式找slot

核心思路 - 8.9s通用版本

- 重放计算第一阶段 - 流水线计算 (7.25s)
  - 单线程顺序读取十个文件，通过协调者分配任务，并行解析出LogOperation(数据操作对象)
  - 通过同步，保证LogOperation的顺序，重放计算者重放出数据库中最后符合主键在查询范围内记录的hashset
  - 重放计算者计算时，会有hashmap来保持当前数据库所有记录信息

- 重放计算第二阶段 – 并行evaluate (0.25s)
  - 先把hashset转化为数组
  - 并行处理每一条记录 (计算出对应的落盘时候的byte[])，把处理结果插入key为主键， value为byte[]的ConcurrentSkipListMap对象中

- 重放计算第二阶段后 - 网络传输和落盘 (0.25s)
  - 遍历有序的 ConcurrentSkipListMap， 产生出结果文件对应的 byte[] ； 使用 java nio 的 transferFrom 方式，将结果文件byte[] 直接发送到Client落盘，这样网络传输和落盘就不会经过用户空间，是最佳的zero-copy的方式

内容

1、赛题理解、核心思路概览

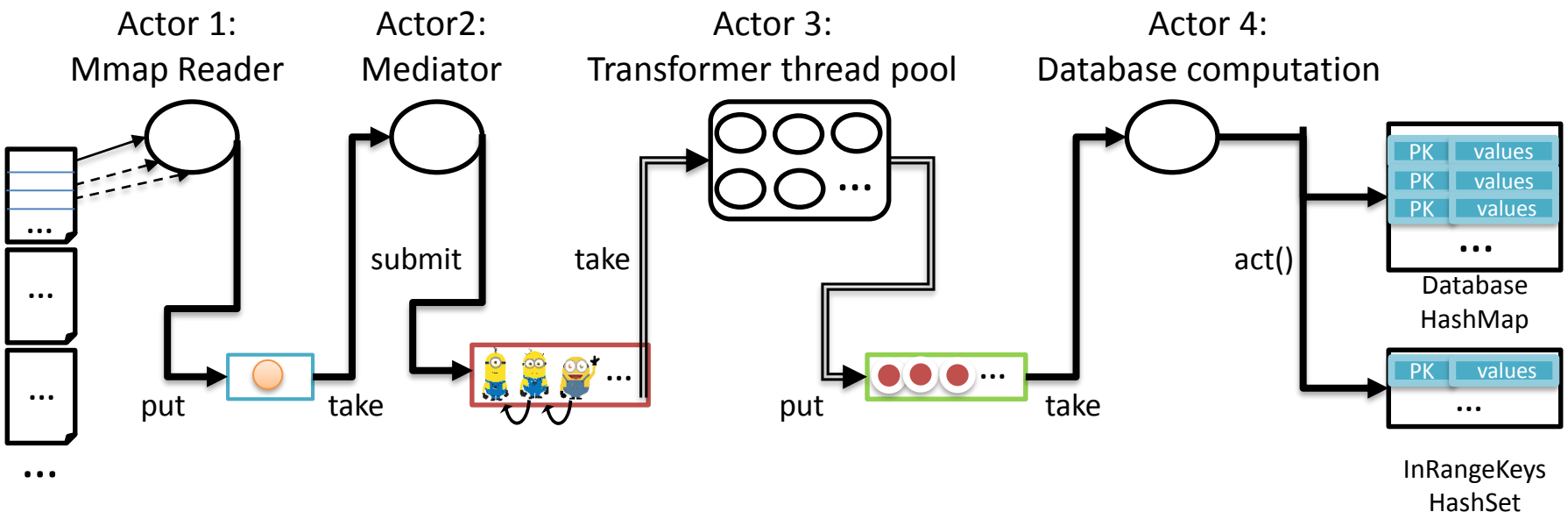
2、重放计算流水线、网络传输落盘设计

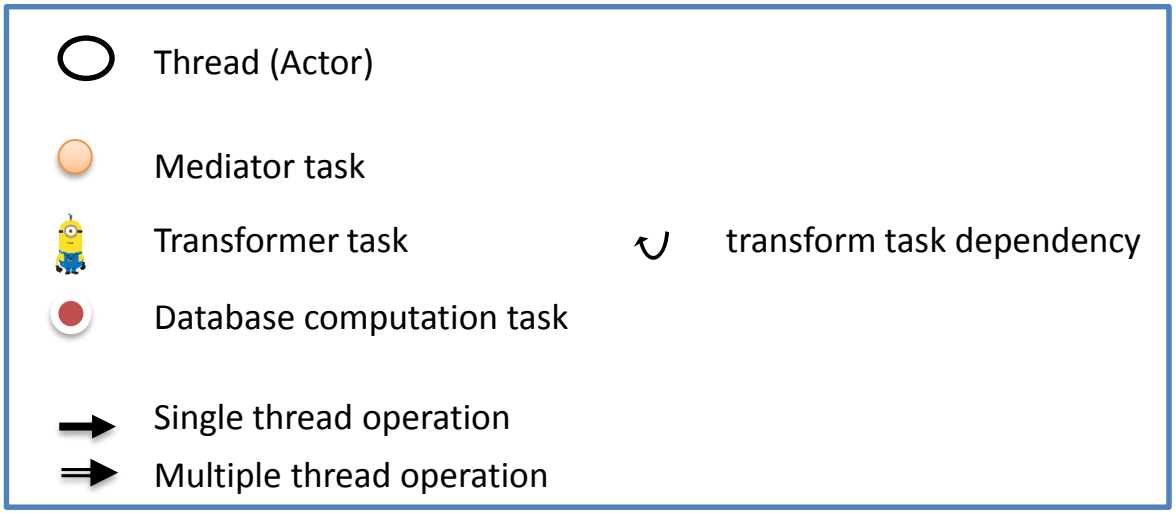
3、工程价值、通用性分析

4、理论最优时间、tricks分析

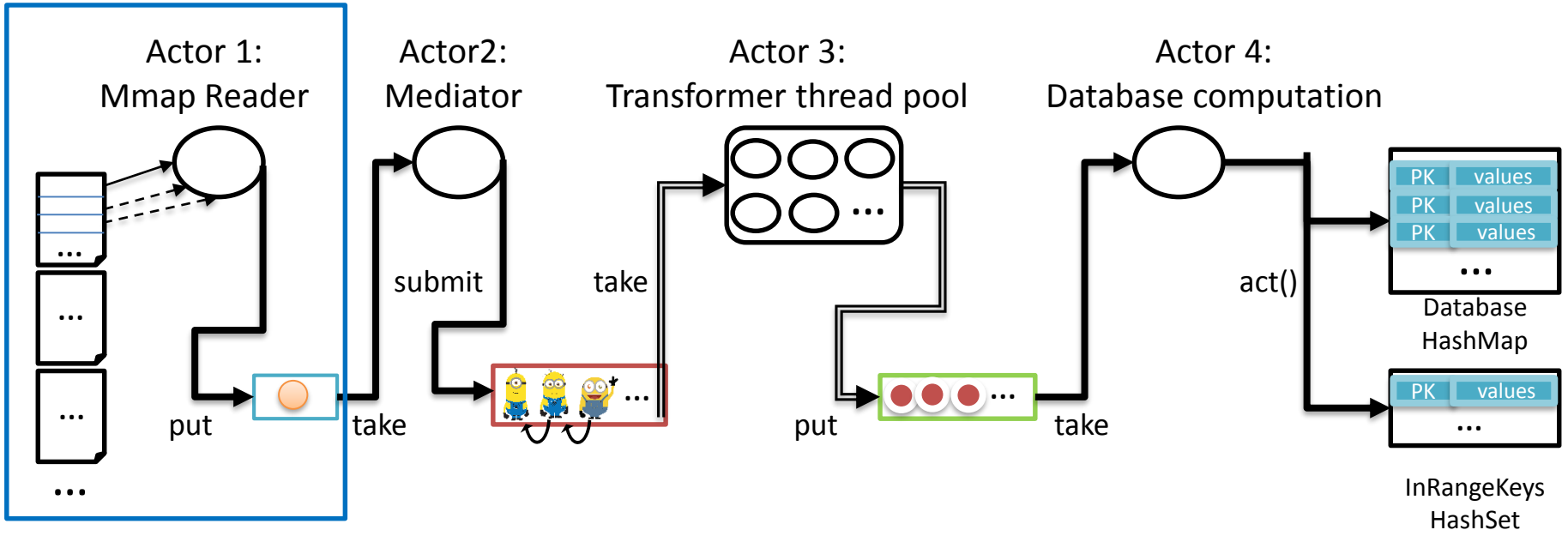
5、版本演进、比赛总结

第一阶段 - 流水线设计

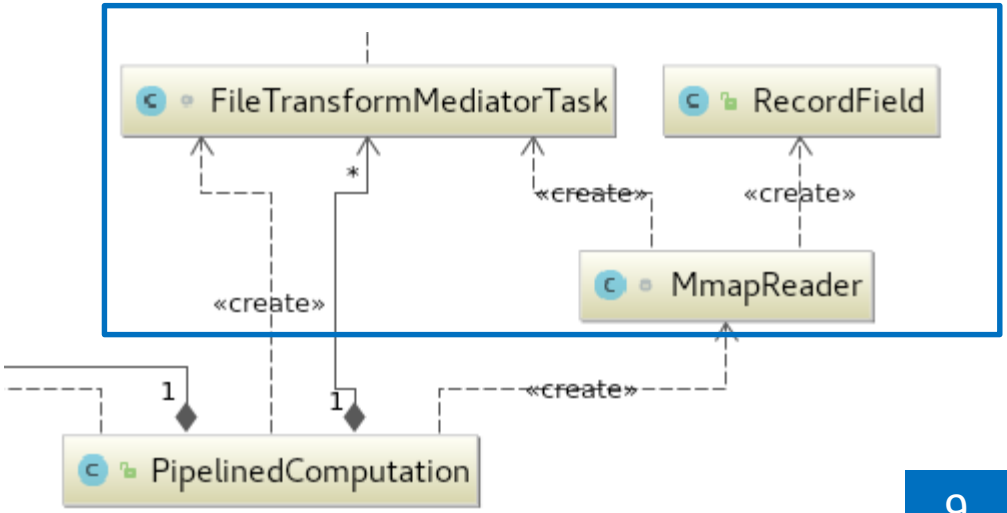

Actor 1:
Mmap Reader
Actor2:
Mediator
Actor 3:
Transformer thread pool
Actor 4:
Database computation
put
take
submit
take
put
take
act()
PK
values
Database
HashMap
InRangeKeys
HashSet


Thread (Actor)
Mediator task
Transformer task
transform task dependency
Database computation task
Single thread operation
Multiple thread operation

Actor 1: MmapReader (主线程)



- 职责
  - 初始化RecordField静态变量
  - 生产FileTransformMediatorTask



MmapReader 职责 – 初始化DDL信息

- 初始化RecordField静态变量：首次读取文件时，初始化单表中的字段相关信息，存入RecordField类静态变量中，为actor 3 transformer解析模块准备

```
public static Map<ByteBuffer, Integer> fieldIndexMap = new HashMap<>();
static int[] fieldSkipLen;
public static int FILED_NUM;
static int KEY_LEN;
```

```
private void fetchNextMmapChunk() throws IOException {
    int currChunkLength = nextIndex != maxIndex ? CHUNK_SIZE : lastChunkLength;

    MappedByteBuffer mappedByteBuffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, nextIndex * CHUNK_SIZE, currCh
    mappedByteBuffer.load();
    if (!RecordField.isInit()) {
        new RecordField(mappedByteBuffer).initFieldIndexMap();
    }

    try {
        mediatorTasks.put(new FileTransformMediatorTask(mappedByteBuffer, currChunkLength));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

MmapReader 职责 – 生产任务

- 生产FileTransformMediatorTask：
  - 生产64MB大小的mmap chunk，放入容量为1的task queue
  - 通过load保证了流式读取的要求

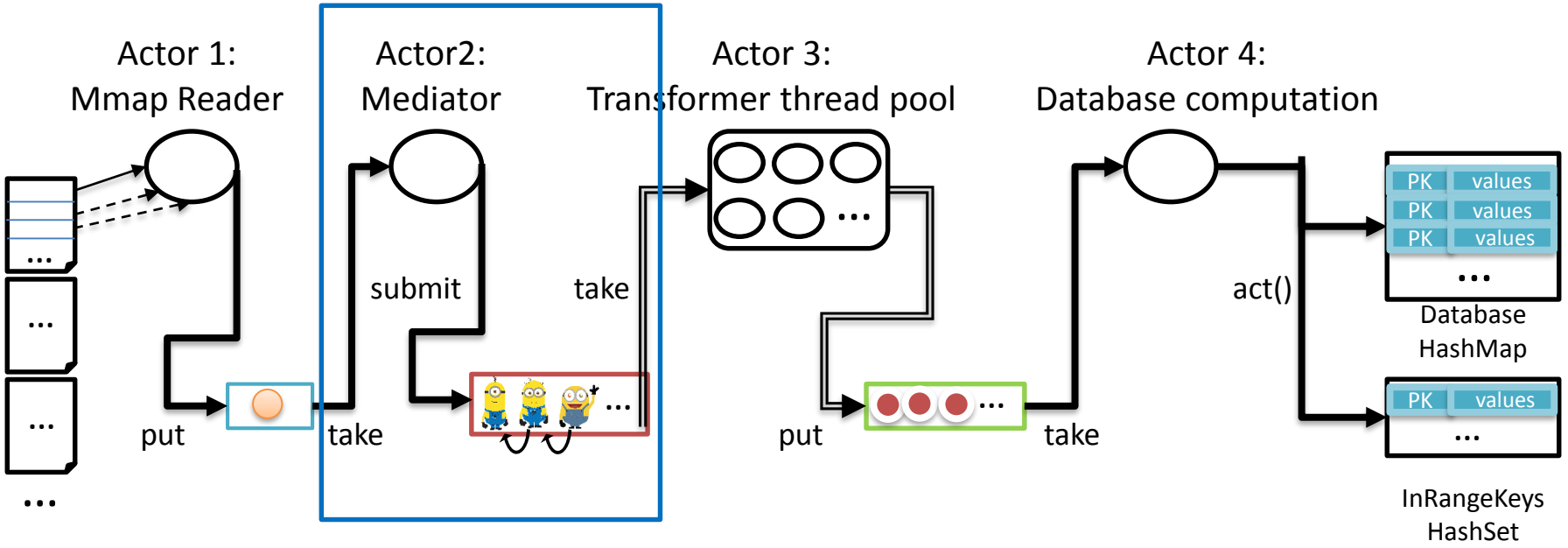
```
private void fetchNextMmapChunk() throws IOException {
    int currChunkLength = nextIndex != maxIndex ? CHUNK_SIZE : lastChunkLength;

    MappedByteBuffer mappedByteBuffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, nextIndex * CHUNK_SIZE, currCh
    mappedByteBuffer.load();
    if (!RecordField.isInit()) {
        new RecordField(mappedByteBuffer).initFieldIndexMap();
    }

    try {
        mediatorTasks.put(new FileTransformMediatorTask(mappedByteBuffer, currChunkLength));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

MmapReader 职责 – 生产任务

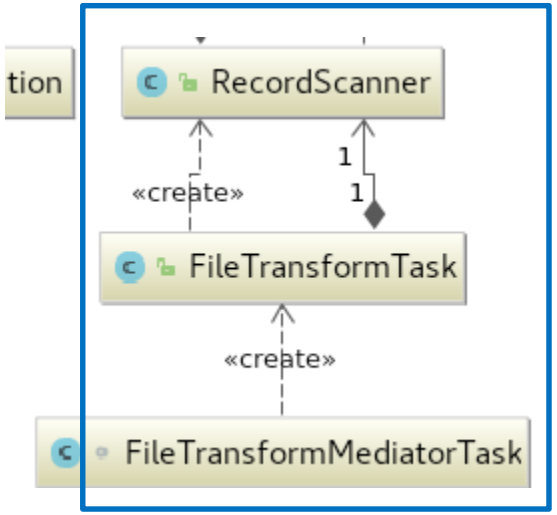
- 生产FileTransformMediatorTask：
  - 生产64MB大小的mmap chunk，放入容量为1的task queue
  - 通过load保证了流式读取的要求
  - 内存中至多存在3份mmap chunk，也就是actor1 mmap reader中1份，actor2 mediator中各1份(mediator中会有同步机制来unmap对应的mmap chunk)，blocking queue中1份，合计192MB)

```
private void fetchNextMmapChunk() throws IOException {
    int currChunkLength = nextIndex != maxIndex ? CHUNK_SIZE : lastChunkLength;

    MappedByteBuffer mappedByteBuffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, nextIndex * CHUNK_SIZE, currCh
    mappedByteBuffer.load();
    if (!RecordField.isInit()) {
        new RecordField(mappedByteBuffer).initFieldIndexMap();
    }

    try {
        mediatorTasks.put(new FileTransformMediatorTask(mappedByteBuffer, currChunkLength));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Actor 2: Mediator (单个线程)



- 职责
  - 轮询获取任务，直到收到完成信号结束
  - 关键协调工作1：分割mmap chunk，计算start index和end index，保证每一个transformer处理完整的记录
  - 关键协调工作2：缓存mmap chunk尾部不足一行Log的byte[]，交给下一轮第一个transformer处理
  - 处理前后任务间依赖



Mediator 职责 - 轮询处理任务

- Mediator轮询获取任务，直到遇到结束信号break

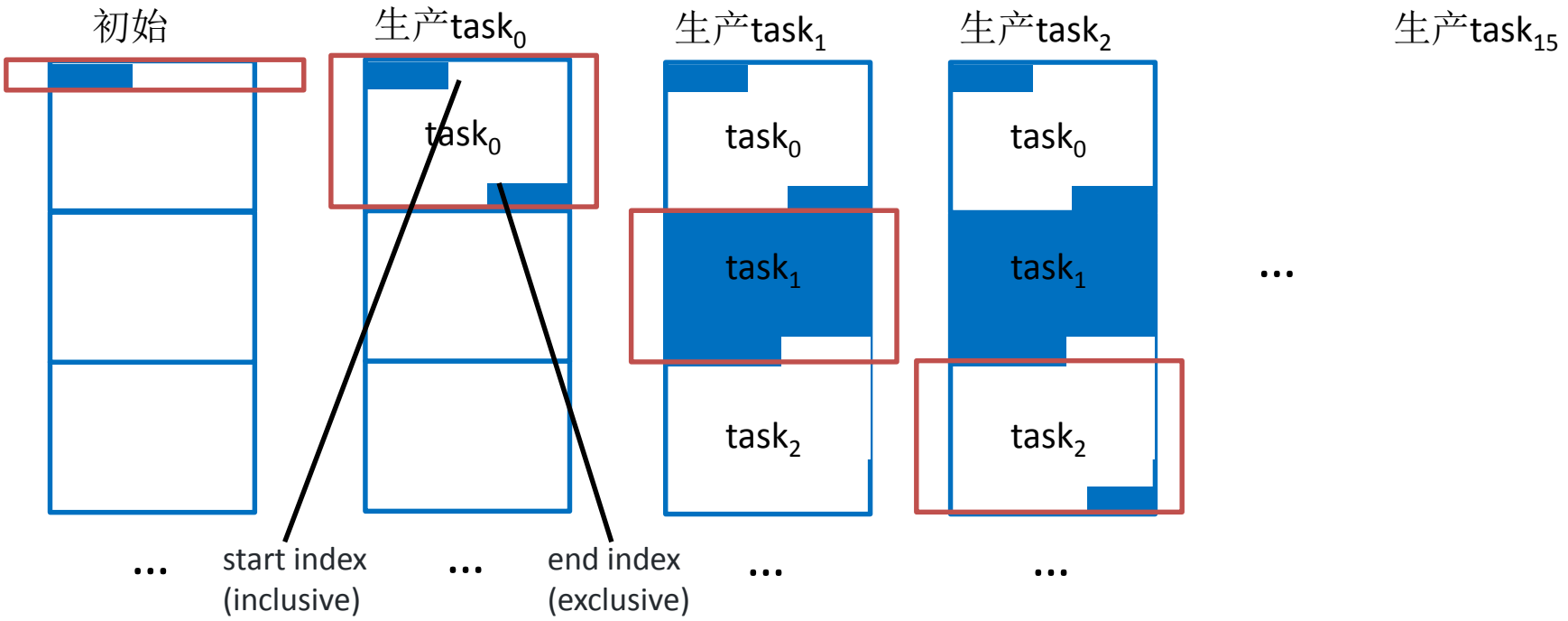
```
mediatorPool.execute(new Runnable() {
    @Override
    public void run() {
        while (true) {
            try {
                FileTransformMediatorTask fileTransformMediatorTask = mediatorTasks.take();
                if (fileTransformMediatorTask.isFinished)
                    break;
                fileTransformMediatorTask.transform();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
});
```

- 主线程在顺序读取完文件后，发送不含内容的Task；其中，isFinished成员变量为true，作为结束信号

```
try {
    mediatorTasks.put(new FileTransformMediatorTask());
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Mediator 职责 – 分配transformer任务

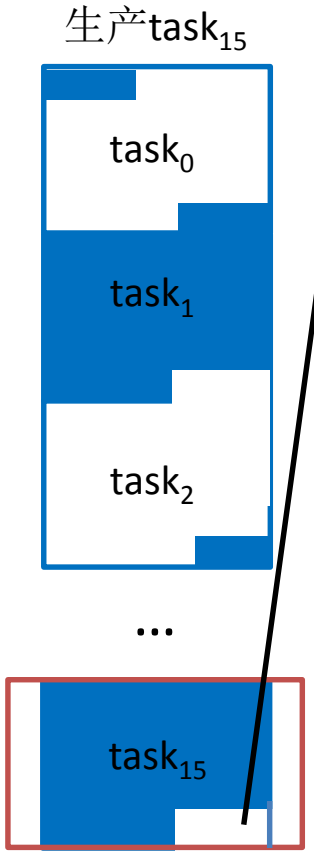
- 把64MB 的 mmap chunk静态分割成16份；保证每一个transformer处理完整的日志行，任务通过start index(inclusive)和end index(exclusive)表示



- 注意点
  - task开始index：在|mysql...的|上
  - task结束index：在\n的后一个上

Mediator 职责 – 处理尾部byte[]

- 静态分割任务的最后一小块，需要特殊处理尾部，在这个例子中是$task_{15}$



- 缓存尾部byte[]，存入prevRemainingBytes

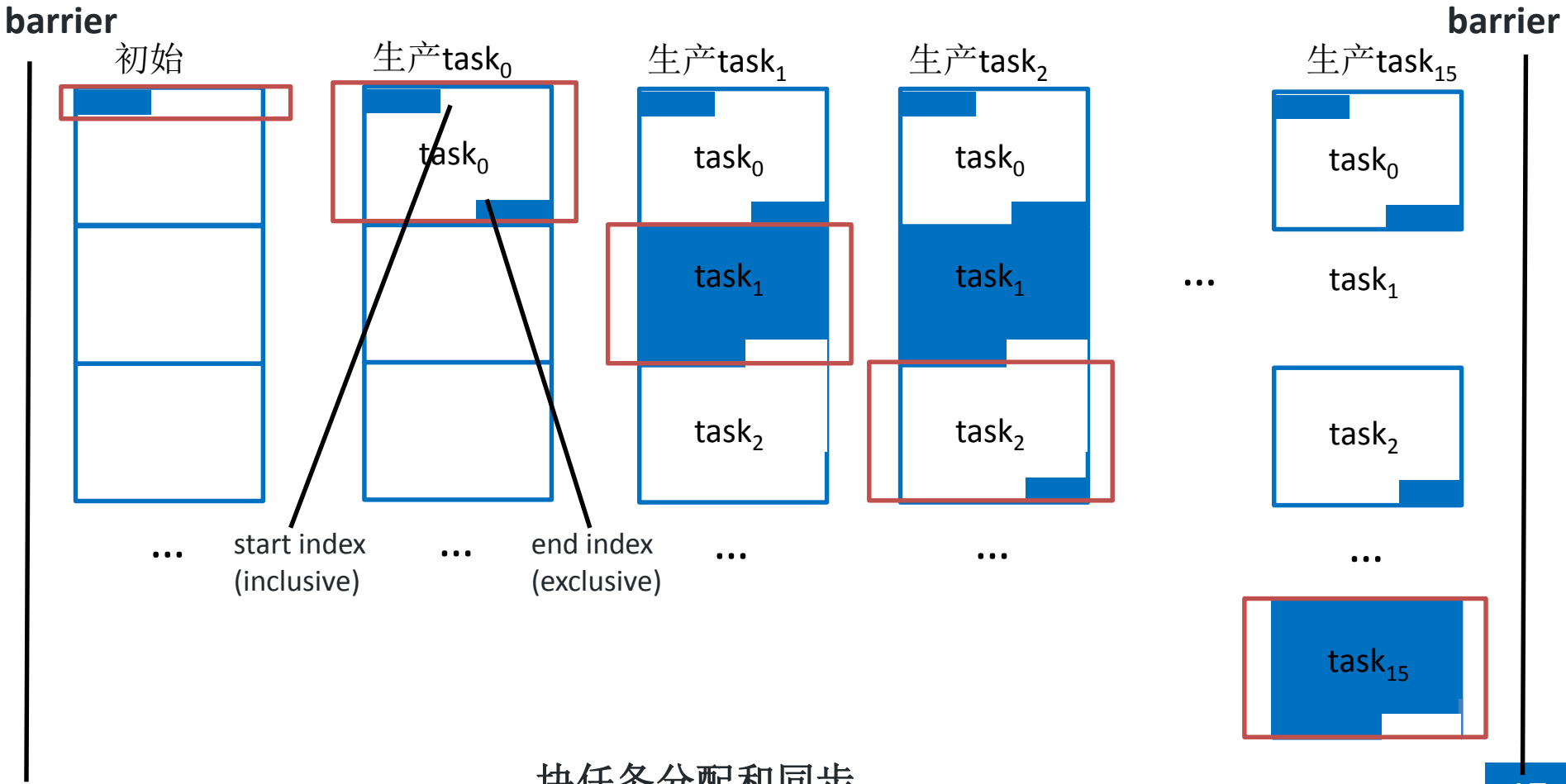
```
prevRemainingBytes.clear();
for (int i = end; i < currChunkLength; i++) {
    prevRemainingBytes.put(mappedByteBuffer.get(i));
}
```

- 下一轮让第一个task额外处理tmp这个ByteBuffer对象

```
// index pair
int start;
int end = preparePrevBytes();

// 1st: first worker
start = end;
end = computeEnd(avgTask - 1);
FileTransformTask fileTransformTask;
if (prevRemainingBytes.limit() > 0) {
    ByteBuffer tmp = ByteBuffer.allocate(prevRemainingBytes.limit());
    tmp.put(prevRemainingBytes);
    fileTransformTask = new FileTransformTask(mappedByteBuffer, start, end, tmp, prevFuture);
} else {
    fileTransformTask = new FileTransformTask(mappedByteBuffer, start, end, prevFuture);
}
```

Mediator 职责 – 块同步，unmap

- submit完16个任务，需要进行同步，等待所有transformer完成，然后unmap，以保证mediator只会持有1份mmap chunk



块任务分配和同步

Mediator 职责 - 处理任务间依赖

- 生产给transformer的任务分为两个环节：1）LogOperation的解析，2）LogOpeartion[]放入重放计算队列；为保证重放时候的顺序性：后一个任务的下半环节需等待前一个任务完成
  - 使用一个类静态变量prevFuture，从中可以获取上一个任务完成状态
  - submit任务时候时候更新prevFuture
  - 创建transformer任务对象时候传入prevFuture

```
private static Future<?> prevFuture = new Future<Object>() {
    @Override
    public boolean isDone() {
        return true;
    }
```

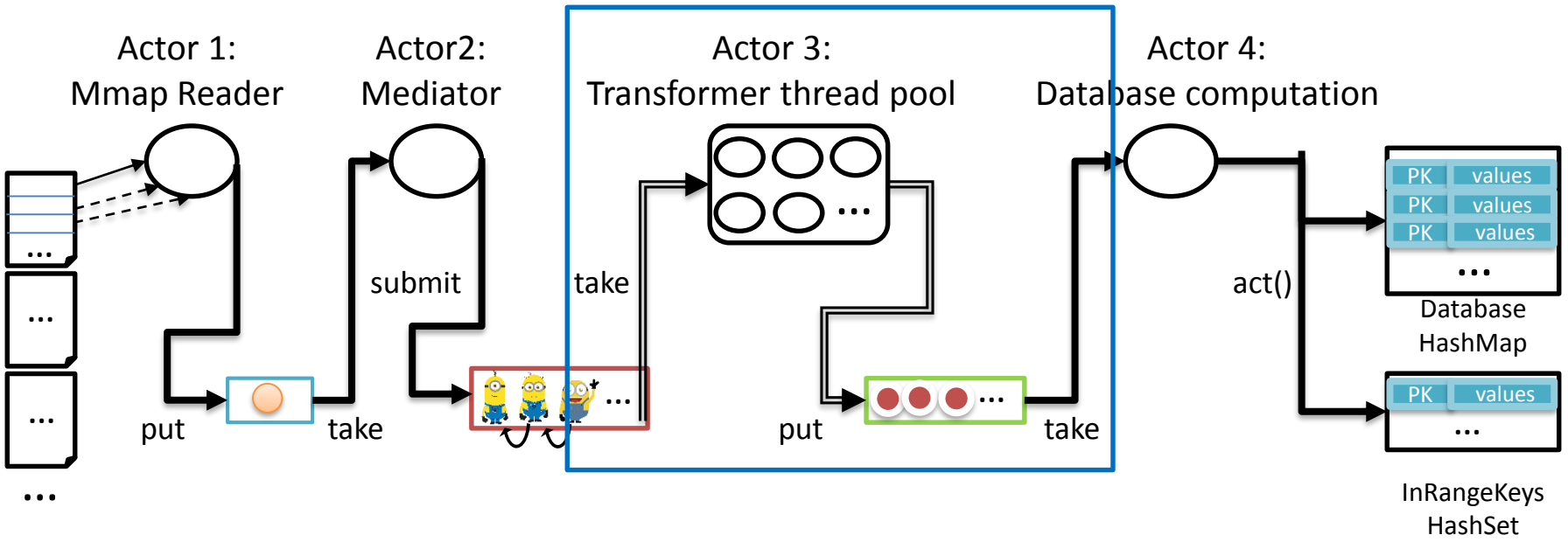
```
prevFuture = fileTransformPool.submit(fileTransformTask);
```

```
fileTransformTask = new FileTransformTask(mappedByteBuffer, start, end, prevFuture);
```

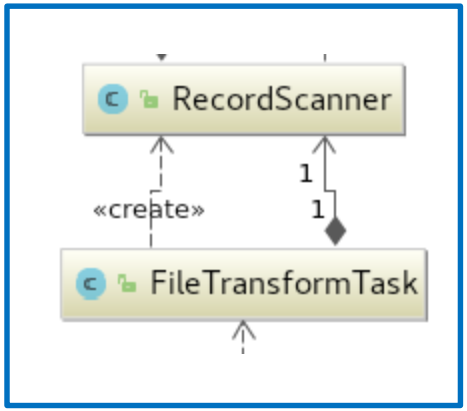
- Transformer任务中的同步逻辑

```
void waitForSend() throws InterruptedException, ExecutionException {
    // wait for producing tasks
    LogOperation[] logOperations = localOperations.toArray(new LogOperation[0]);
    prevFuture.get();
    PipelinedComputation.blockingQueue.put(logOperations);
}
```

Actor 3: Transformer Pool (16线程)

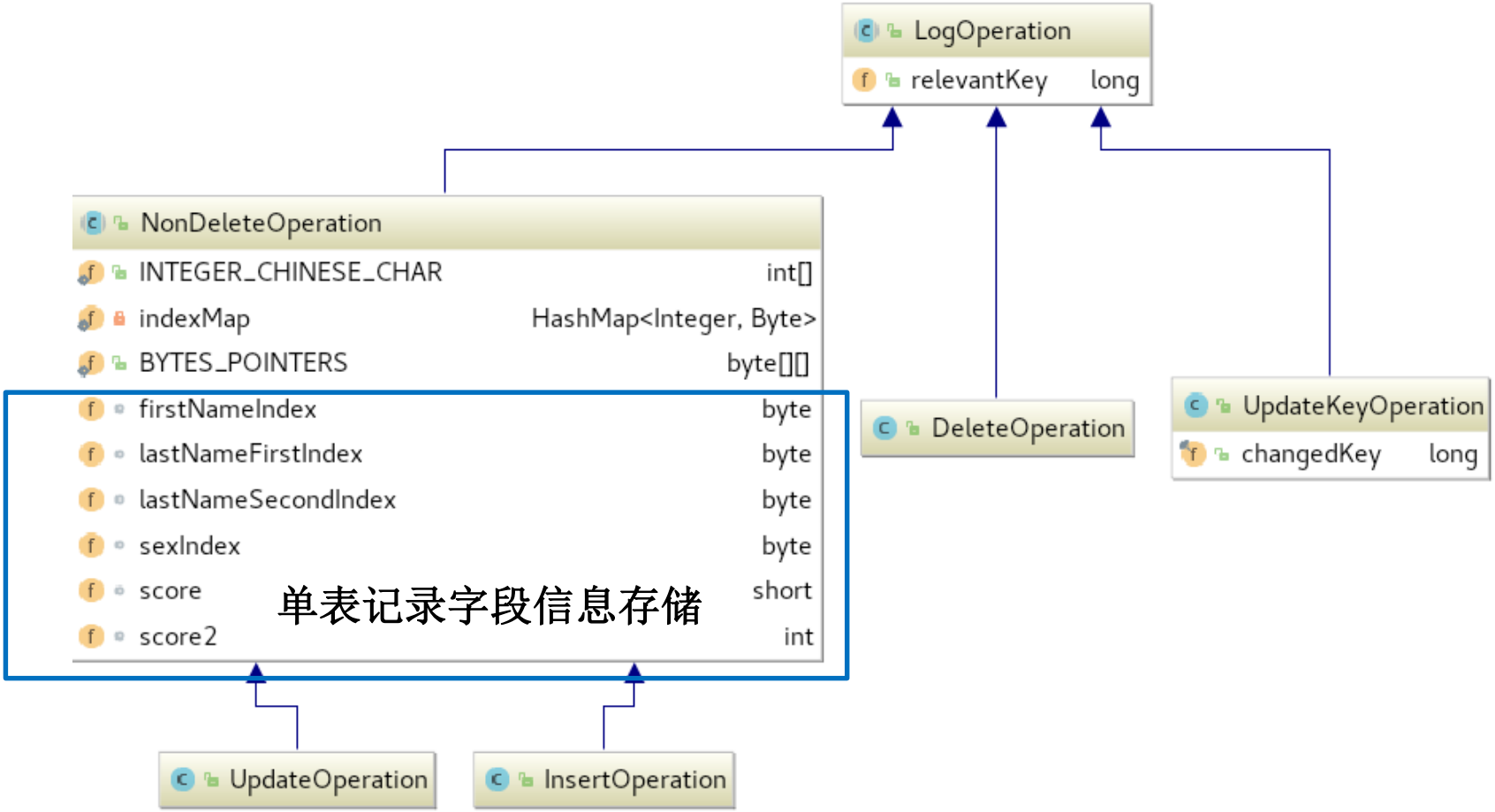


- 职责
  - 解析：将对应完整的日志行chunk，解析为数据操作对象数组(LogOperation[])
  - 同步：等待上一个任务完成生产，也就是把LogOperation[]放入重放计算的task queue中
  - 生产：将LogOperation[]放入重放计算的task queue中



解析出的LogOperation类型

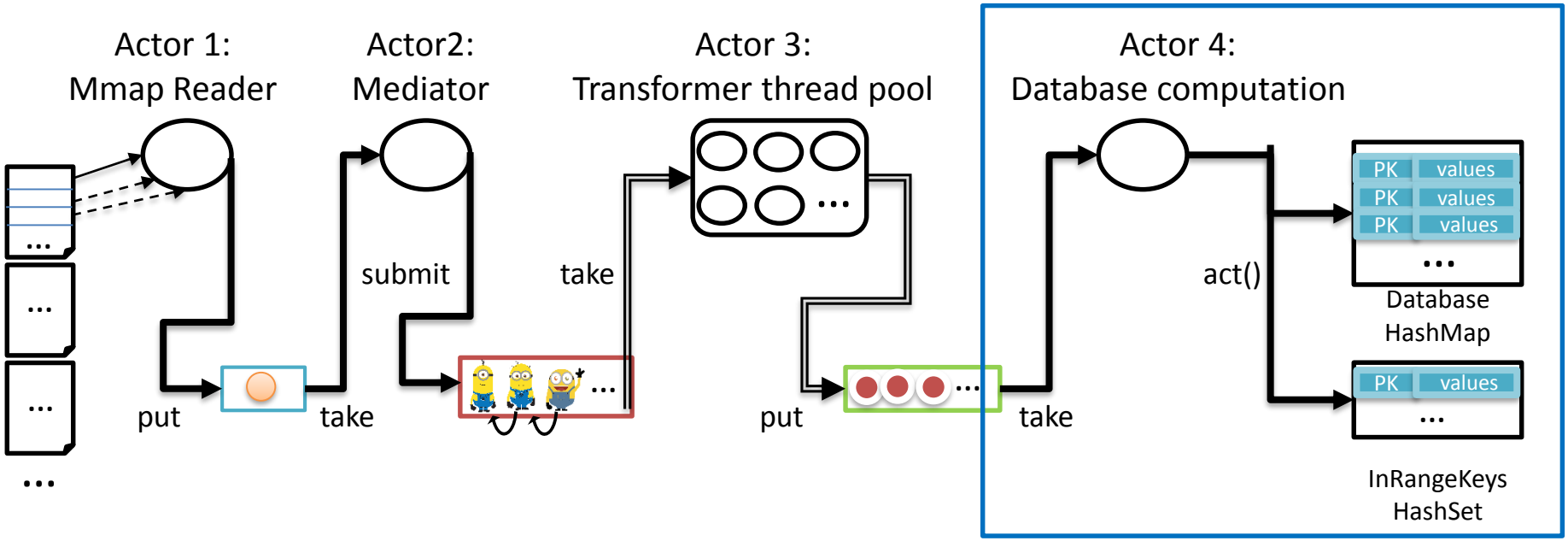
- LogOperation类图



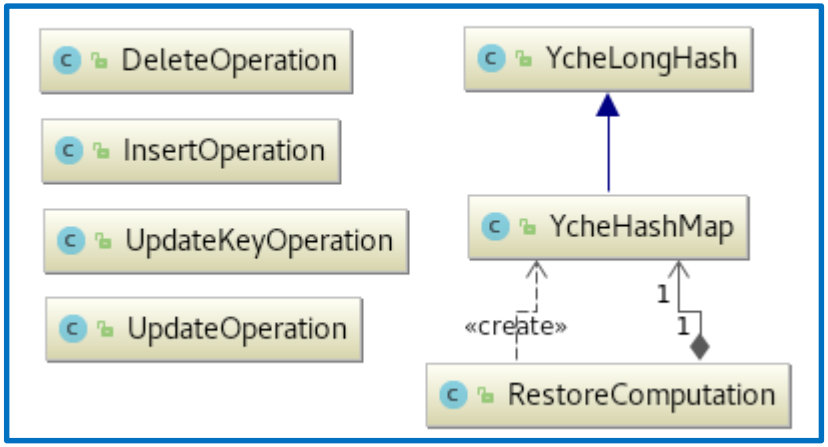
LogOperation特点和生命周期

- InsertOperation
  - 包含记录所有属性
  - 存活时间长，仅仅当被删除时候或Update主键时候死亡
  - 最后在查询范围内的InsertOperation对象被用来evaluate出对应的byte[]
- DeleteOperation
  - 快速消费
- UpdateOperation
  - 包含一个有效变更属性
  - 快速消费
- UpdateKeyOpeartion
  - 包含一个变更前主键，一个变更后主键
  - 快速消费

Actor 4: Computation Worker (单个线程)



- 职责
  - 轮询处理任务(逻辑类似Mediator轮询)
  - 重放数据库
  - 维护查询范围内记录集合



Actor 4: Computation Worker (单个线程)

- 数据结构
  - recordMap(key为long/value为LogOpeartion，参考trove hashmap改写)：当前数据库中，所有记录 (不执行删除操作是为了去掉hashmap中状态数组，并减少查hashmap次数)
  - inRangeRecordSet：存储在范围内的记录
  - 注意：存储的LogOpeartion类型必定为InsertOperation

```
public static YcheHashMap recordMap = new YcheHashMap(24 * 1024 * 1024);
public static THashSet<LogOperation> inRangeRecordSet = new THashSet<>(4 * 1024 * 1024);
```

- 数据库重放计算
  - LogOperation的时序性，由actor 3 transformer解析出LogOperation[]后，放入computation task queue时的同步机制保证
  - 对LogOperation采取对应的行为，更新recordMap和inRangeRecordSet

```
static void compute(LogOperation[] logOperations) {
    for (LogOperation logOperation : logOperations) {
        logOperation.act();
    }
}
```

重放计算 (Delete, Insert)

- DeleteOperation
  - inRangeRecordSet：若DeleteOperation在range范围内，就把其从inRangeRecordSet中delete
  - recordMap：不变

```
@Override
public void act() {
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }
}
```

- InsertOperation
  - inRangeRecordSet：若InsertOperation在range范围内，就把其加入到inRangeRecordSet中
  - recordMap：直接insert

```
@Override
public void act(){
    recordMap.put(this); //1
    if (PipelinedComputation.isKeyInRange(relevantKey)) {
        inRangeRecordSet.add(this);
    }
}
```

重放计算 (Update Property, Update Key)

- UpdateOperation
  - inRangeRecordSet : 不变，因为普通属性变更不会影响主键
  - recordMap：首先probing获取InsertOperation对象，然后进行insertOperation.mergeAnother(this)，落实这次属性的变更

```
@Override
public void act(){
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this); //2
    insertOperation.mergeAnother(this); //3
};
```

- UpdateKeyOperation
  - 等效于进行了一次DeleteOperation，再接着进行一次InsertOperation
  - 注意点：InsertOperation中，会带来有原来被删除的对应记录中的所有属性
  - 注意点举例：UpdateKeyOperation进行了1->3的主键变更，那么我们需要首先获取所有主键1对应记录的属性，然后delete主键1对应记录，然后insert主键3，并且主键3对应记录包含之前主键1中获取到的所有属性

重放计算 (Update Key)

- UpdateKeyOperation
  - recordMap: 取出Update Key之前，对应记录

```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this); //2
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }

    insertOperation.changePK(this.changedKey); //4
    recordMap.put(insertOperation); //5

    if (PipelinedComputation.isKeyInRange(insertOperation.relevantKey)) {
        inRangeRecordSet.add(insertOperation);
    }
}
```

重放计算 (Update Key)

- UpdateKeyOperation
  - recordMap: 取出Update Key之前，对应记录
  - inRangeRecordSet：如果这个变更之前的key在range内，把其从删除

```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this); //2
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }

    insertOperation.changePK(this.changedKey); //4
    recordMap.put(insertOperation); //5

    if (PipelinedComputation.isKeyInRange(insertOperation.relevantKey)) {
        inRangeRecordSet.add(insertOperation);
    }
}
```

重放计算 (Update Key)

- UpdateKeyOperation
  - recordMap: 取出Update Key之前，对应记录
  - inRangeRecordSet：如果这个变更之前的key在range内，把其从删除
  - recordMap：通过insertOperation.changePK(this.changedKey)，得出新key对应对象，insert该新对象

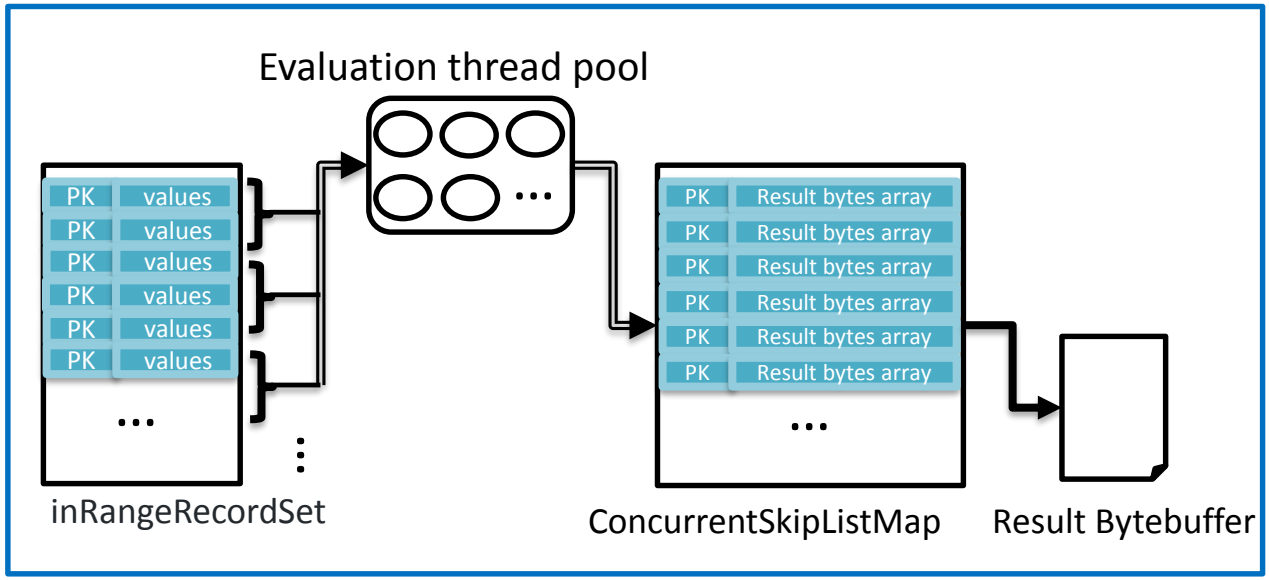
```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this); //2
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }

    insertOperation.changePK(this.changedKey); //4
    recordMap.put(insertOperation); //5

    if (PipelinedComputation.isKeyInRange(insertOperation.relevantKey)) {
        inRangeRecordSet.add(insertOperation);
    }
}
```

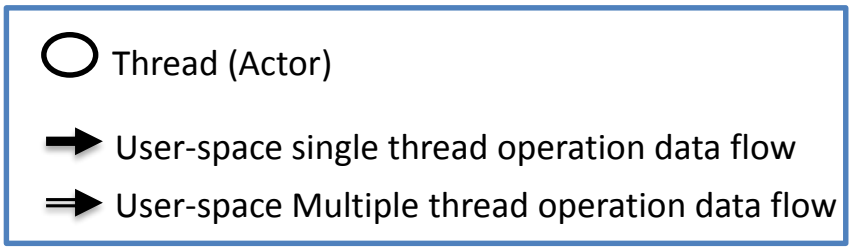
重放计算 (Update Key)

- UpdateKeyOperation
  - recordMap: 取出Update Key之前，对应记录
  - inRangeRecordSet：如果这个变更之前的key在range内，把其从删除
  - recordMap：通过insertOperation.changePK(this.changedKey)，得出新key对应对象，insert该新对象
  - inRangeRecordSet: 如果key若在range内，把其加入

```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this); //2
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }

    insertOperation.changePK(this.changedKey); //4
    recordMap.put(insertOperation); //5

    if (PipelinedComputation.isKeyInRange(insertOperation.relevantKey)) {
        inRangeRecordSet.add(insertOperation);
    }
}
```

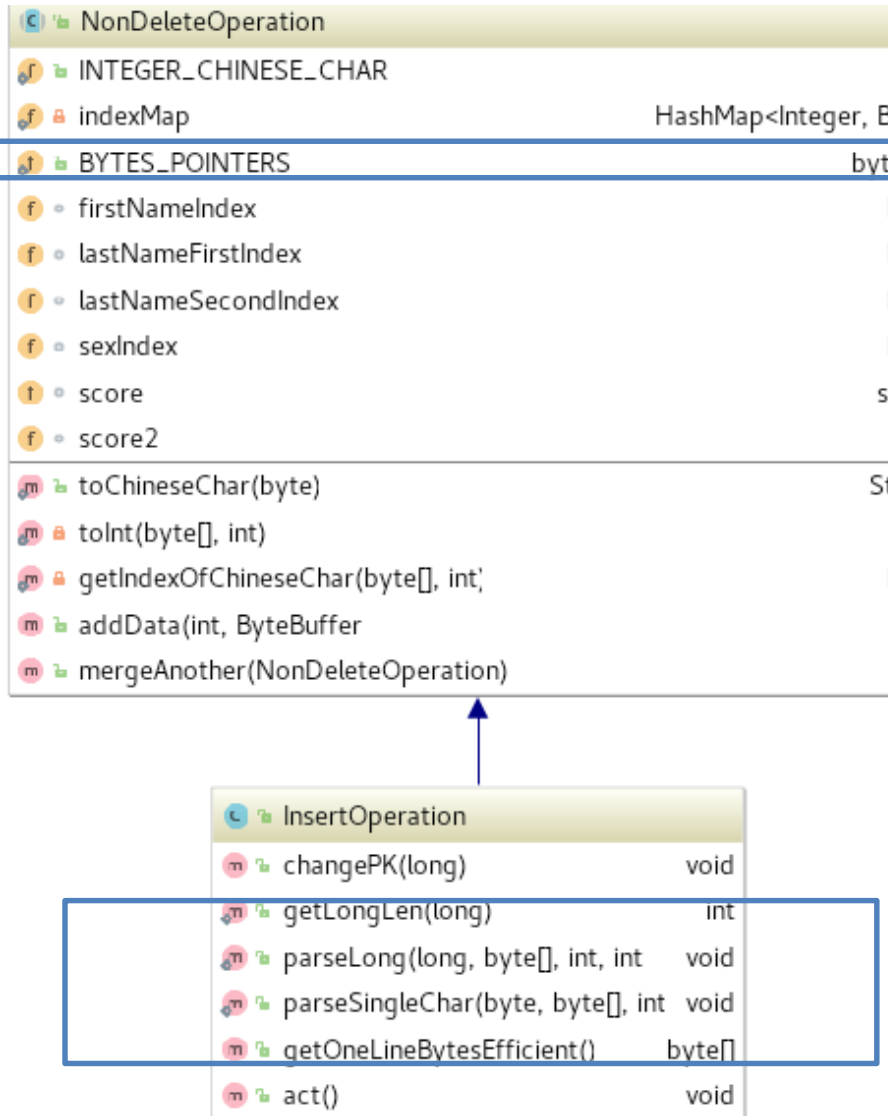
第二阶段 - 并行evaluate (16线程)





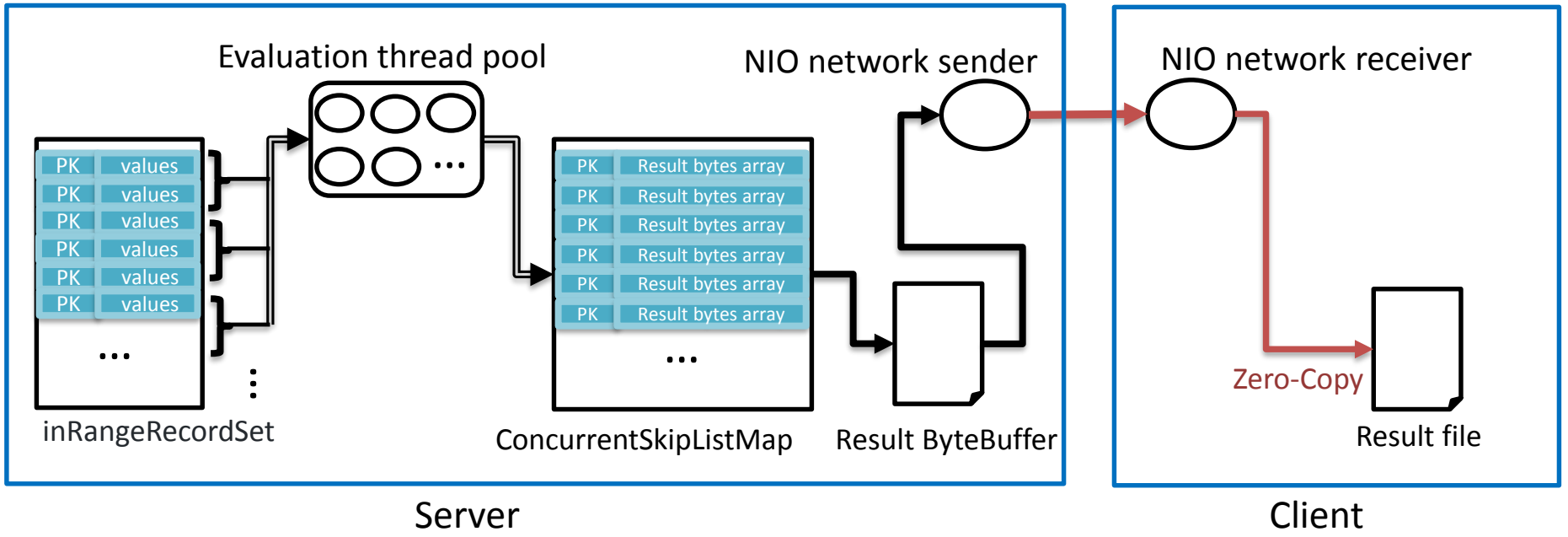
- 注意点
  - 任务分配前，先通过toArray()，把inRangeRecordSet转为数组
  - 任务分配时，直接分配begin Index和end index
  - 并行eval后，遍历skiplistmap，构建结果ByteBuffer

第二阶段 – evaluate getOneLineBytesEfficient



- 注意点
  - 之前：用StringBuilder实现会比较慢，因为StringBuilder创建和append调用。
  - 调优：用了直接的byte[]的操作，和自己写的转换函数parseLong和parseSingleChar
  - 提升：0.5s cost减到 0.25s

第二阶段后 - Zero-Copy 网络传输落盘



Thread (Actor)

User-space single thread operation data flow

User-space Multiple thread operation data flow

Kernel-space data flow

- 注意点
  - 网络传输和落盘都不经过用户态空间

内容

1、赛题理解、核心思路概览

2、重放计算流水线、网络传输落盘设计

3、工程价值、通用性分析

4、理论最优时间、tricks分析

5、版本演进、比赛总结

工程背景的契合度

- 简洁的流水线设计解耦了mmap reader/mediator/transformer/computation worker，提高了模块的重用性和扩展性；扩展时候只要对相应模块小部分逻辑更改
  - mmap reader：顺序mmap并load，10G文件保证了流式处理，符合了只能单线程顺序读取所有文件内容一遍的要求
  - mediator：负责解耦transform完后放入计算队列的任务依赖，比较简单
  - Transformer：相关处理逻辑，在通用化时候，只需要修改RecordScanner和NonDeleteOperation中小部分内容
  - computation worker：重放逻辑的实现支持在不修改表DDL信息前提下，数据操作任何变更下的正确性

- 扩展到实际场景：作为canal后续模块，多表多范围对应多client重放的可能性
  - 流水线的主体逻辑不需要大变动
  - 不同表对应不同的RecordScanner和NonDeleteOperation
  - computation worker的数据结构可以变为 (1个表，1个范围) 对应 (1个hashmap，1个hashset)

健壮性 1 – 不同数据集适应性

- 重放计算数据结构回顾
  - recordMap(key为long/value为LogOpeartion，参考trove hashmap改写)：当前数据库中，所有记录(不执行删除操作是为了去掉hashmap中状态数组，并减少查hashmap次数)
  - inRangeRecordSet：存储在范围内的记录
  - 注意：存储的LogOpeartion类型必定为InsertOperation

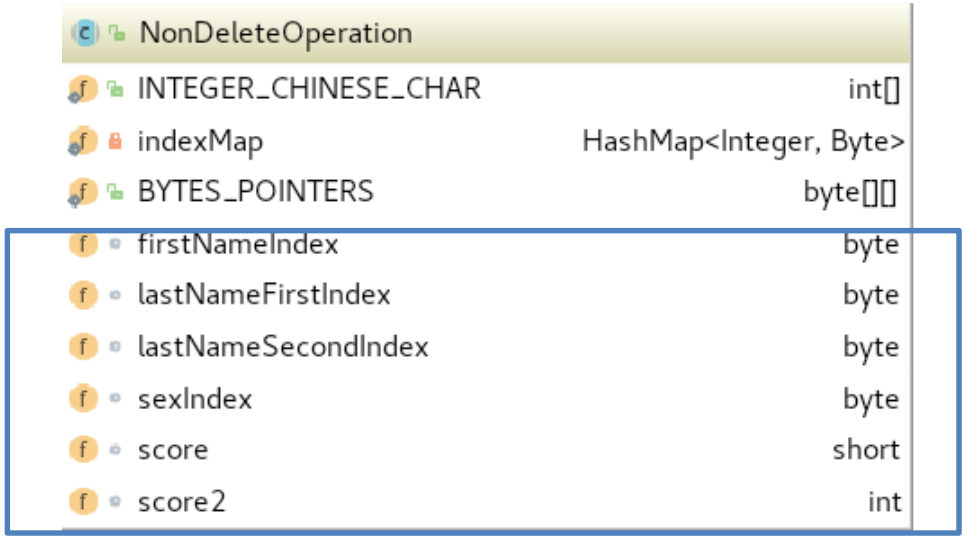
```
public static YcheHashMap recordMap = new YcheHashMap(24 * 1024 * 1024);
public static THashSet<LogOperation> inRangeRecordSet = new THashSet<>(4 * 1024 * 1024);
```

- 针对不同数据操作的通用性设计，可以适应下面情形
  - 基于单表的，含有不同DML操作的不同的数据集
  - 主键在不同范围，例如含有2^{63}

健壮性 2 – 针对不同表通用化

- 观察：数据库中一般表中的字段信息和字段长度范围是确定的
- 修改：对应的字段，不同表对应不同的NonDeleteOperation；对应的对象构建时候addData函数，update落实mergeAnother函数

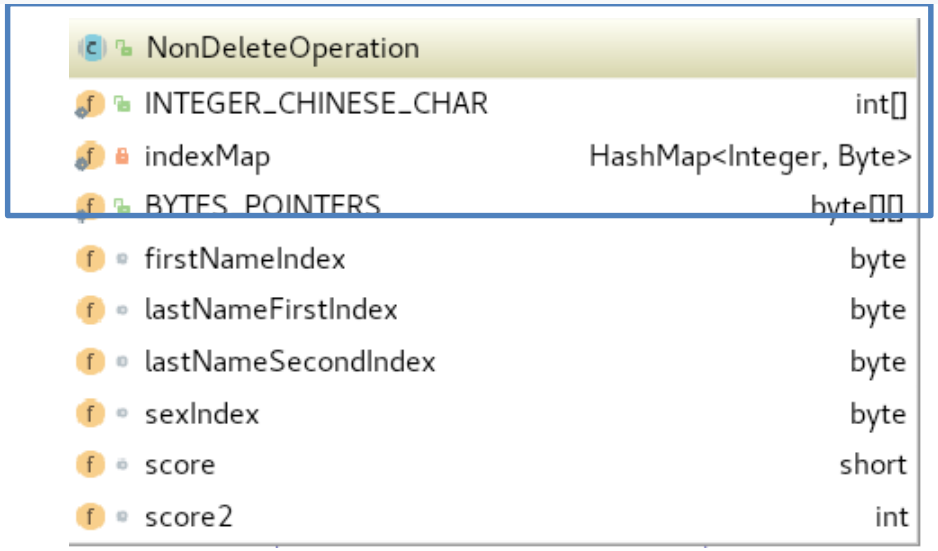
```
public void addData(int index, ByteBuffer byteBuffer) {
    switch (index) {
        case 0:
            firstNameIndex = getIndexOfChineseChar(byteBuffer.array(), 0);
            break;
public void mergeAnother(NonDeleteOperation nonDeleteOperation) {
    if (nonDeleteOperation.score != -1) {
        this.score = nonDeleteOperation.score;
        return;
    }
```



健壮性 2 – 针对不同表通用化

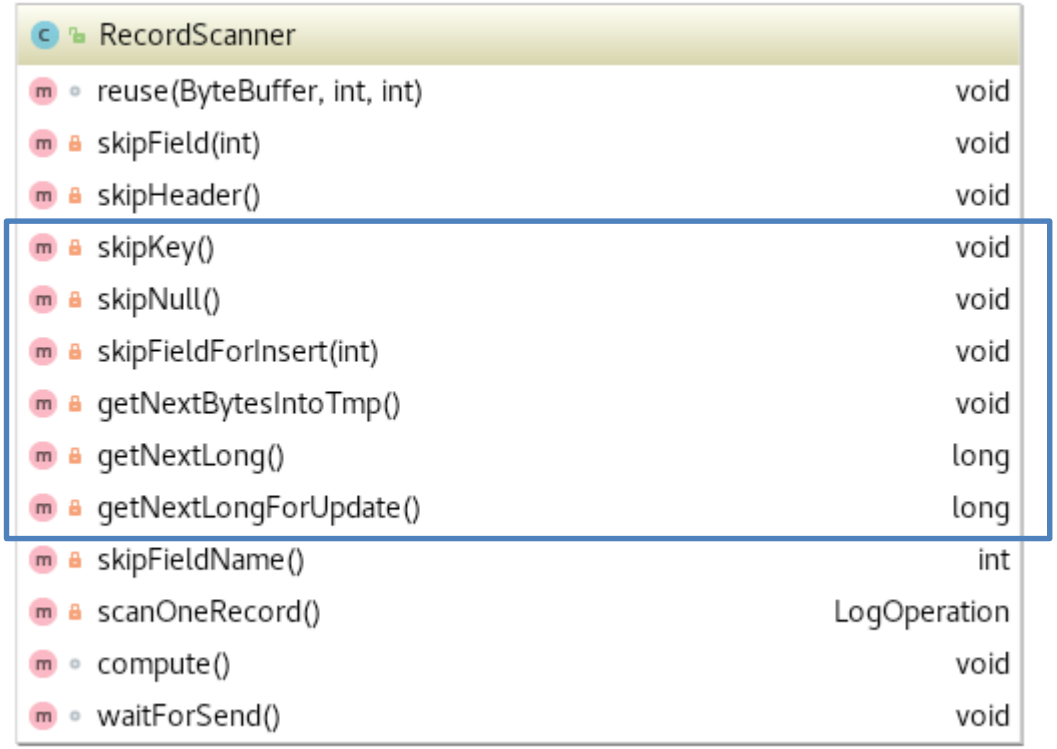
- 观察：数据库中一般表中的字符对应可能表示内容是确定的，有限的
- 修改：初始化时候初始化对应可能的内容

```
public static int[] INTEGER_CHINESE_CHAR = {14989440, 14989441, 14989443, 14989449, 14989450, 14989465, 14989712, 149
private static HashMap<Integer, Byte> indexMap = new HashMap<>();
public static byte[][] BYTES_POINTERS = new byte[INTEGER_CHINESE_CHAR.length][];

static {
    for (byte i = 0; i < INTEGER_CHINESE_CHAR.length; i++) {
        indexMap.put(INTEGER_CHINESE_CHAR[i], i);
        BYTES_POINTERS[i] = InsertOperation.toChineseChar(i).getBytes();
    }
}
```



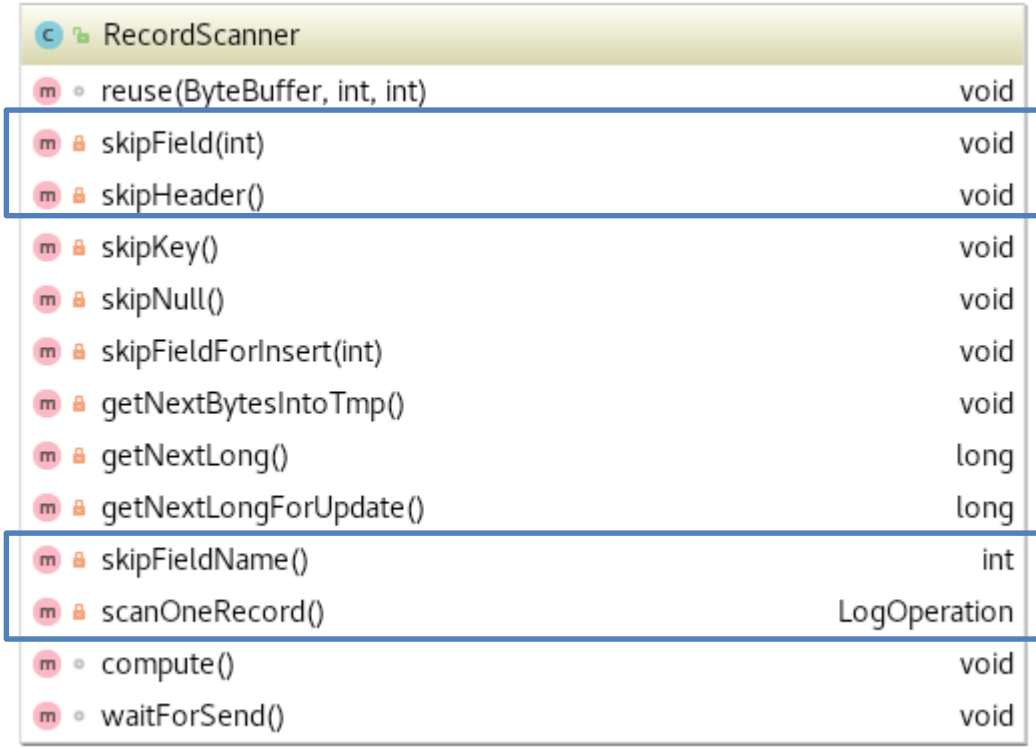
健壮性 2 – 解析逻辑

- 不需要修改的函数
  - 基于DDL解析：skipKey()，skipFieldForInsert(int)
  - 基于canal输出文件特点：skipNull()
  - 与表DDL信息无关的解析函数：getNextBytesIntoTmp()，getNextLong()，getNextLongForUpdate()



健壮性 2 – 解析逻辑

- 需要修改的函数
  - 根据header(mysql-binlog/timestamp/schema/table)：skipHeader()
  - 根据表字段范围：skipField(int)
  - update/delete时候根据字段的头直接判断出字段内容：skipFieldName()
  - 解析逻辑修改：scanOneRecord()



内容

1、赛题理解、核心思路概览

2、重放计算流水线、网络传输落盘设计

3、工程价值、通用性分析

4、理论最优时间、tricks分析

5、版本演进、比赛总结

Trick 1: 利用Update Key特征

- 规律：主键变更，并不需要考虑带来原来主键的属性
- 规律举例
  - 主键从1->3，主键1原来属性一定会被全update；或者变更后最后不在range范围内，不需要取出其属性
- 规律应用
  - update key的操作就可以简单变成两个操作，一个delete之前主键，另一个insert新的主键(不置有任何属性)
  - 只要keep在范围内的主键相关记录，只有1000000到8000000的key对应记录有用，不会出现2^{63}的key有用；所以才可以使用array模拟hashmap表示对应的数据库，array下标对应key, 引用对应value
- 使用后效果
  - 从通用版本8.9s减少到到应用trick1后版本6.7s
  - 第一阶段流水线消耗时间从7.25s减少到5.00s，有显著提升
  - 其中有2.5s是无法避免的mmap load 10G文件开销，也就是说理论最优第一阶段流水线IO和处理及计算完全overlap消耗时间是2.5s，overlap效果还不错

Trick 1: 利用Update Key特征

- 数据结构

```
public static LogOperation[] ycheArr = new LogOperation[8 * 1024 * 1024];
```

- 数据操作 - DeleteOperation

```
@Override
public void act() {
    ycheArr[(int) (this.relevantKey)] = null;
}
```

- 数据操作 - InsertOperation

```
@Override
public void act() {
    ycheArr[(int) (this.relevantKey)] = this;
}
```

- 数据操作 - UpdateOperation

```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) RestoreComputation.ycheArr[(int) (this.relevantKey)]; //2
    if(insertOperation==null){
        insertOperation=new InsertOperation(this.relevantKey);
        RestoreComputation.ycheArr[(int) this.relevantKey]=insertOperation;
    }
    insertOperation.mergeAnother(this); //3
}
```

用于在下面一页PPT分析的trick2

Trick 2: 选择真正有用的文件chunk计算

- 想法：基于trick1的规律，其实可以通过一次提交，统计出最后真正有用的文件小chunk(以4MB为单位)，mediator只submit有用的小chunk对应的任务

- 统计内容
  - 最终被delete掉的主键，最后一次delete操作对应文件小chunk的global index
  - 最终在查询范围内数据的各个属性对应文件小chunk的global index
- 统计内容汇总
  - 对上面得出小chunk全局index，做set_union操作

- 目的：大大减少transformer的任务和重放计算的任务

- 使用后效果
  - 从trick1版本6.7s减少到到应用trick2后版本4.8s
  - 真正执行transform和computation的小chunk个数为508个，原来总数2688个
  - 第一阶段流水线消耗时间从5.00s减少到3.25s，有显著提升
  - 其中有2.5s是无法避免的mmap load 10G文件开销，也就是说理论上最优的第一阶段流水线IO和处理及计算完全overlap消耗时间是2.5s，已经比较接近理论极限

Trick 3: mmap后不load (我们团队未使用)

- 本地尝试：基于trick2，改一行代码去掉load ， MmapReader mmap后不load
- 本地效果
  - 本机8逻辑核CPU运行总时间从3.50s减少为1.50s
  - 本机与线上环境类似，只是没用网络传输，用的热身赛数据
- 预期线上效果
  - 线上程序应该也可以缩小到1.50s左右，加上评测程序和jvm启动开销1.00s，这个版本提交到线上，成绩应该可以从4.80s减少为2.50s
- 为什么不在线上提交
  - 违反规则，没有将所有文件内容读取一遍，只mmap不load，相当于跳过了文件某些部分的读取
  - 如果这一点都违反了，那么倒着读或者并发读都可以做了，因为性质相同，都属于不遵循**单线程顺序读取所有文件内容1次**这一要求

内容

1、赛题理解、核心思路概览

2、重放计算流水线、网络传输落盘设计

3、工程价值、通用性分析

4、理论最优时间、tricks分析

5、版本演进、比赛总结

版本演进 - 通用性代码

日期	成绩	版本说明	commit
06/19	59.268s	放弃拷贝文件并且倒着读思路，实现正着重放版本，属性存储使用byte[][]，记录和操作用不同小对象存储	90b9f3a9
06/19	46.163s	使用InsertOperation直接存储对应记录中的属性信息，减少内存拷贝和额外的小对象创建	dae43787
06/19	43.433s	优化RecordScanner中getNextLong()；依据单表字段信息不变特征，添加skipFieldForInsert(int index)	9d29966d
06/21	39.265s	使用ArrayBlockingQueue来协调生产者给消费者发任务，消费者使用轮询方式来获取任务	1ac36f81a
06/22	20.155s	依据单表中字段长度范围固定，优化RecordScanner和InsertOperation	e8a6389c
06/23	14.456s	重放计算中，采用访存更加友好的gnu trove hashmap改写版本	f5811f02
06/24	10.867s	优化网络传输和落盘，实现Zero-Copy；优化并行evaluate模块	2576b8b5
06/26	8.979s	优化if-else分支，使用多态，使用gnu trove hashset	e631f265

版本演进 - 采用Tricks(最后两天)

日期	成绩	版本说明	commit
06/27	7.906s	利用Update Key的trick，主键变更不会带来之前属性	2a956e52
06/28	6.670s	利用Update Key trick，使用数组代替hashmap和hashset	36bb463f
06/29	4.819s	使用终极trick，计算时候跳过不需要的文件chunk, 并且不使用logger，但是还是在MmapReader读文件时候调用了load，保证了不违反规则	b31990db

比赛总结和思考

- 初赛
  - 学习了pagecache，只有把文件读写的size压缩到pagecache乘vm.dirty_ratio的大小才可以取得比较好的性能
  - 学习到了一些快速的压缩算法，例如snappy和lz4
- 复赛
  - 明白了访存pattern的重要性，通过byte[][]的方式会慢，而通过扁平化的存储会块；认识到了在java中，默认的extends Object会带来将近8byte的开销，并且jvm会进行8byte内存对齐，在设计小对象的时候要格外小心
  - 学习了用来overlap计算和IO的并行计算流水线的设计，解决了一些相关的同步和并发控制问题，来取得比较好的性能
  - hashmap和hashset的具体实现影响性能，jdk自带的基于拉链的组织开销会比较大，开地址方式实现时hashmap的probing方式对于性能影响也很大
- 感想
  - 注意操作系统和语言底层vm实现相关的内容
  - 从阿里中间件博客学习到了许多新知识，之后要多多关注业界的技术热点

结束

初赛：https://github.com/CheYulin/OpenMessageShaping

复赛：https://github.com/CheYulin/IncrementalSyncShaping

比赛攻略：https://github.com/CheYulin/IncrementalSyncShaping/tree/master/comp_summary